

Communication interprocessus

Hassen JEDIDI

Plan

1. Introduction
2. Sections critiques et exclusion mutuelle
3. Exclusion mutuelle par attente active
 1. Le masquage des interruptions
 2. Les variables de verrouillage
 3. L'alternance stricte
 4. La solution de *Peterson*
4. Exclusion mutuelle sans attente active
 - a. Les primitives *sleep* et *wakeup*
 - b. Les sémaphores
 - c. Les moniteurs



Introduction

Sur une plateforme multiprogrammée les processus ont généralement besoin de communiquer pour compléter leurs tâches.

L'exécution d'un processus peut être affecter par l'exécution des autres processus ou il peut affecter lui-même leurs exécutions.

La communication interprocessus est assurée généralement via des données partagées qui peuvent se trouver dans la mémoire principale ou dans un fichier.



Les accès concurrents (simultanés) à des données partagées peuvent conduire à des incohérences dans les résultats obtenus.



Introduction: exemple illustratif

Exemple : la spoule d'impression

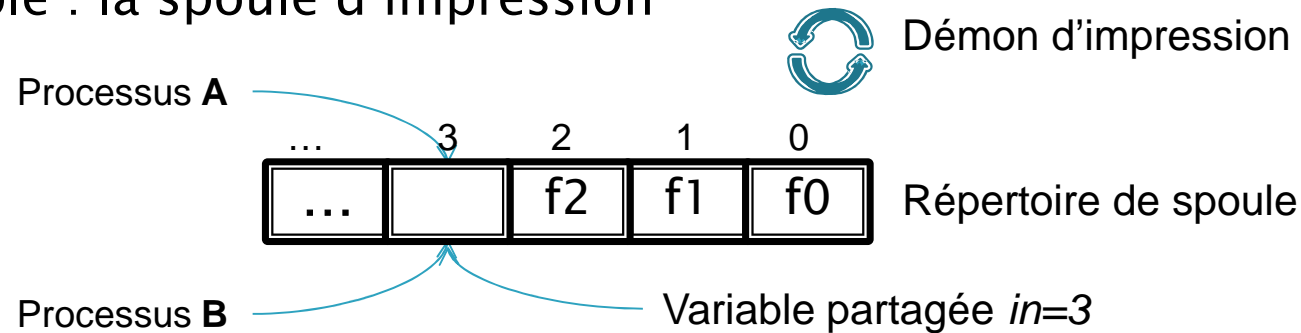
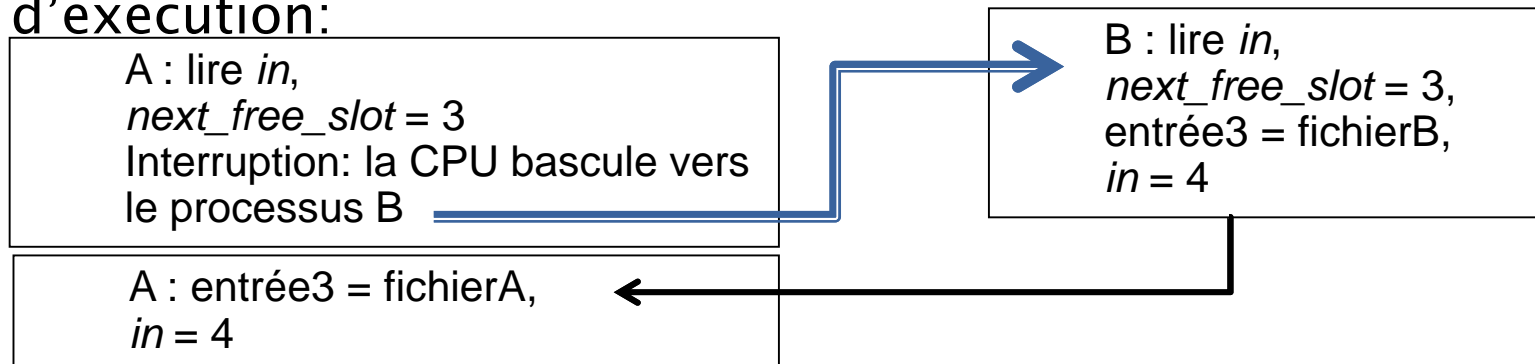


Schéma
d'exécution:



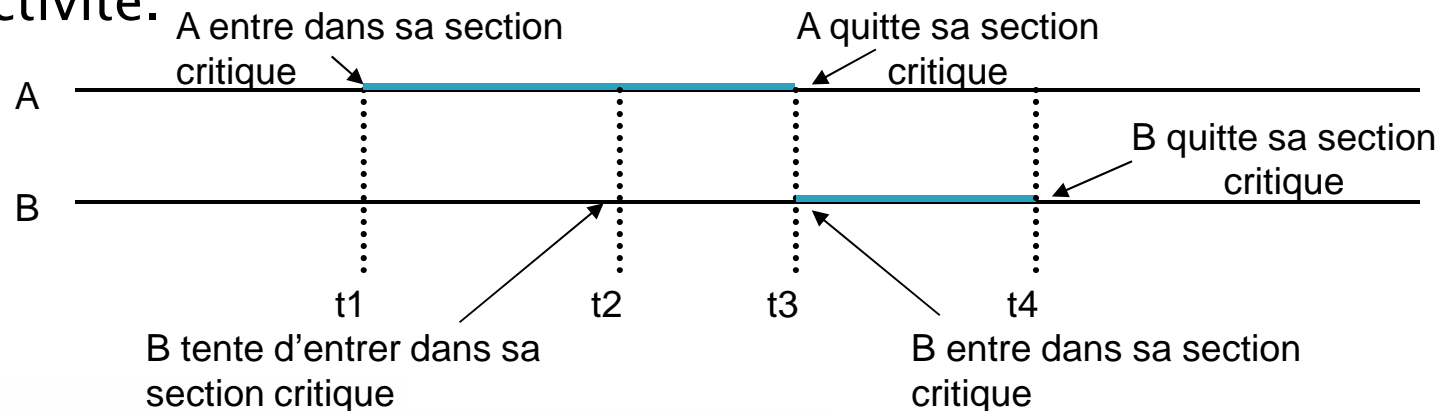
Problème: le fichier B ne sera jamais imprimé

Sections critiques et exclusion mutuelle

Le problème précédent est dû aux conflits d'accès à la même ressource.

La partie du programme à partir de laquelle on accède à la ressource partagée est appelée **section (région) critique**.

Solution: **L'exclusion mutuelle** est une méthode qui assure qu'un seul processus est autorisé d'accéder à une ressource partagée; les autres processus seront exclus de la même activité.



Sections critiques et exclusion mutuelle

Quatre conditions doivent être vérifiées pour assurer l'exclusion mutuelle:

1. Deux processus ne doivent pas se trouver simultanément dans leurs sections critiques.
2. Aucun processus à l'extérieur de sa section critique ne doit bloquer les autres processus.
3. Aucun processus ne doit attendre indéfiniment pour entrer dans sa section critique.
4. Il ne faut pas faire d'hypothèse quant à la vitesse ou le nombre de processeurs



Exclusion mutuelle par attente active

Un processus désirant entrer dans une section critique doit être mis en attente si la section critique devient libre.

Un processus quittant la section critique doit le signaler aux autres processus.

Algorithme d'accès à une section critique :

```
Entrer_Section_Critique ()    /* attente si SC non libre */  
Section_Critique()           /* un seul processus en SC */  
Quitter_Section_Critique()
```

L'attente peut être :

- **Active** : la procédure *Entrer_Section_Critique* est une boucle dont la condition est un test qui porte sur des variables indiquant la présence ou non d'un processus en Section critique.
- **Non active** : le processus passe dans l'état endormi et ne sera réveillé que lorsqu'il sera autorisé à entrer en section critique.

Solutions de l'exclusion mutuelle par attente active

Solution 1: Masquage des interruptions

- Lorsqu'un processus entre en section critique il doit masquer les interruptions.

➡ Pas de commutation de processus

- Lorsqu'il quitte sa section critique il doit restaurer les interruptions.

C'est une solution matérielle qui permet de résoudre complètement le problème. Mais elle est dangereuse en mode utilisateur s'il oublie de restaurer les interruptions.

Solutions de l'exclusion mutuelle par attente active

Solution 2: Variables de verrouillage

Un verrou est une variable binaire **partagée** qui indique la présence d'un processus en section critique.

si verrou=0 alors section critique libre

si verrou=1 alors section critique occupée

```
void entrer_Section_Critique ()  
{  
    while (verrou == 1) ; /* attente active */  
    verrou=1 ;  
}
```

```
Void quitter_Section_Critique ()  
{  
    verrou=0 ;  
}
```

Cette solution ne garantit pas l'exclusion mutuelle car le verrou est une variable partagée qui peut constituer aussi une section critique.

Solutions de l'exclusion mutuelle par attente active

Solution 3: Alternance stricte

Tour est une variable **partagée** qui indique le numéro de processus autorisé à entrer en section critique.

```
void entrer_Section_Critique (int process)
{
    while (Tour!=process) ; /* attente active */
}
```

```
Void quitter_Section_Critique ()
{
    Tour = (Tour+1) ;
}
```

L'alternance stricte est une solution simple et facile à implémenter.

Mais, un processus qui possède Tour peut ne pas être intéressé immédiatement par la section critique et en même temps il bloque un autre processus qui demandeur.

 **Problème de famine**

Solutions de l'exclusion mutuelle par attente active

Solution 4: Solution de Peterson

```
#define FAUX 0
#define VRAI 1
#define N 2
int tour ; /* à qui le tour */
int interesse[N] ; /* initialisé à FAUX */
void entrer_Section_Critique (int process)
{
    int autre ;
    autre = 1-process ;
    interesse[process]=VRAI; /* process est intéressé */
    tour = process ;          /* demander le tour */
    while (tour == process && interesse[autre] == VRAI) ;
}
```

```
Void quitter_Section_Critique ()
{
    interesse[process]=FAUX ;
}
```

Cette solution assure complètement l'exclusion mutuelle.

Mais, le processus qui attend sa section critique consomme du temps processeur inutilement (attente active).

Exclusion mutuelle sans attente active

L'idée est qu'un processus qui ne peut pas entrer en section critique passe à l'état bloqué au lieu de consommer le temps processeur inutilement. Il sera réveillé lorsqu'il pourra y entrer.

Les primitives Sleep et Wakeup:

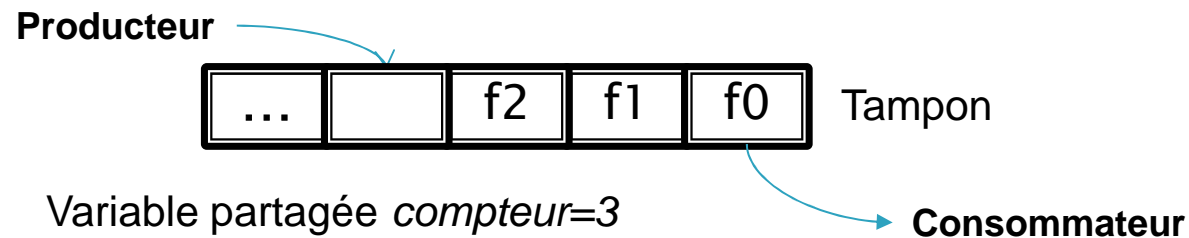
Le système d'exploitation offre deux appels système:

1. Sleep (dormir) qui bloque le processus appelant.
2. Wakeup (réveiller) qui réveille le processus donné en argument.



Exclusion mutuelle sans attente active

Application des primitives Sleep et Wakeup au modèle Producteur Consommateur:



Deux processus (le producteur et le consommateur) coopèrent en partageant un même tampon:

- Le producteur produit des objets qu'il dépose dans le tampon.
- Le consommateur retire des objets du tampon pour les consommer.

Exclusion mutuelle sans attente active

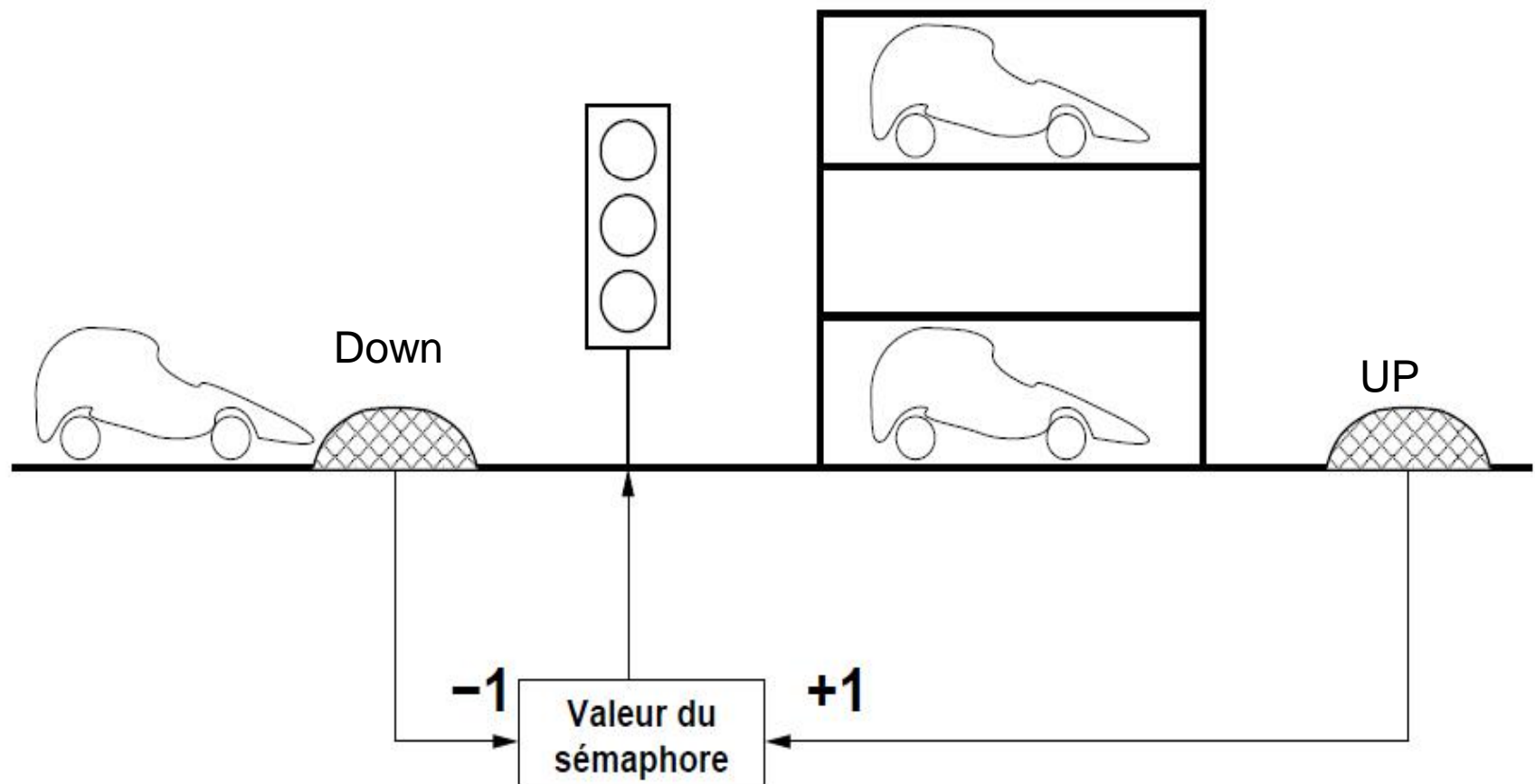
```
#define N 100 /* taille du tampon */
int compteur = 0 ; /* objets dans tampon */
void producteur () {
    while (TRUE)
    {
        produire_objet() ;
        if (compteur == N) sleep () ;
        mettre_objet() ;
        compteur = compteur + 1 ;
        if (compteur == 1)
            wakeup(conconsommateur) ;
    }
}
```

```
void consommeur () {
    while (TRUE)
    {
        if (compteur == 0) sleep() ;
        retirer_objet()
        compteur = compteur - 1 ;
        if (compteur == N-1)
            wakeup (producteur) ;
        consommer_objet(...) ;
    }
}
```

Exclusion mutuelle sans attente active

- Problème de blocage:
 - Le consommateur note que le tampon est vide
 - Interruption: arrêt du consommateur sans qu'il parte en sommeil
 - Le producteur insère un jeton, incrémente le décompte, appelle *wakeup* pour réveiller le consommateur
 - Le signal *wakeup* est perdu, car le consommateur n'est pas en sommeil
 - Le consommateur reprend, pour lui le tampon est vide, il dort
 - Le producteur remplit le tampon et dort
- Solution: ajouter un *bit d'attente d'éveil*.
 - Quand un *wakeup* est envoyé à un processus le bit est à 1;
 - le consommateur teste le bit, s'il est à 1, il le remet à 0 et reste en éveil

Analogie



Les sémaphores

Pour remédier au problème des réveils en attente (les wakeup perdus), l'idée est d'employer une variable entière appelée: **Sémaphore** à laquelle est associée une file d'attente des processus bloqués.

sémaphore=0 → aucun réveil n'est mémorisé
sémaphore>0 → un ou plusieurs réveils sont en attente

Un sémaphore **s** est manipulé par les opérations :

1. **down(s)** : - décrémente la valeur de s si $s > 0$,
- si $s = 0$, alors le processus est mis en attente.
2. **up(s)** : - incrémente la valeur de s,
- si un ou plusieurs processus sont en attente sur ce sémaphore, l'un d'entre eux est réveillé,

Les sémaphores

Pour assurer l'exclusion mutuelle un sémaphore peut être programmé de la manière suivante :

initialisation `mutex = 1` /* nombre de processus autorisés à entrer
simultanément dans la section critique */

Nom du sémaphore



`down (mutex)`

`<section_critique>`

`up (mutex)`

Initialisation d'un sémaphore

```
function Init(semaphore sem, int val)
{
    disable_interrupt;
    sem.K:= val;
    enable_interrupt;
}
```

Opération Down(sem)

```
▶ function Down(semaphore sem)
{ disable_interrupt;
  if (sem.K == 0)
  { L.suivant = processus_courant;
    processus_courant.state= bloque;
    reordonnancement = vrai; }
sem.K=sem.K-1;
enable_interrupt;
}
```

Opération Up(sem)

```
▶ function Up(semaphore sem) {  
    disable_interrupt;  
    sem.K=sem.K+1;  
    if (not L.vide) {  
        processus_reveille= L.tete;  
        processus_reveille.state = prêt,  
        reordonnancement = vrai; }  
    enable_interrupt;  
}
```

Les sémaphores

Application au modèle Producteur / Consommateur :

Trois sémaphores sont nécessaires:

1. **plein**: compte le nombre de places occupées
2. **vide** : compte le nombre de places libres
3. **Mutex** : assure que le producteur et le consommateur n'accèdent jamais en même moment à la mémoire tampon.



Les sémaphores

Application au modèle Producteur / Consommateur :

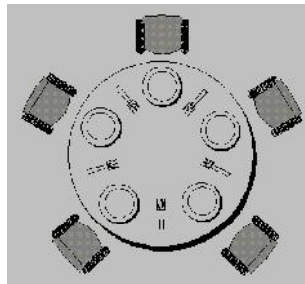
```
#define N 100 /* taille du tampon */  
typedef int semaphore; /* les sémaphores sont des entiers */  
semaphore mutex=1 ; /* contrôle d'accès à la section critique */  
semaphore vide=N; /* contrôle les emplacements vide dans le tampon */  
Semaphore plein=0; /* contrôle les emplacements plein dans le tampon */
```

```
void producteur () {  
while (TRUE)  
    { produire_objet() ;  
      down(vide);  
      down(mutex);  
      mettre_objet() ;  
      up(mutex);  
      up(plein) }  
}
```

```
void consommateur () {  
while (TRUE)  
    { down(plein);  
      down(mutex);  
      retirer_objet()  
      up(mutex);  
      up(vide);  
      consommer_objet(...) ; }  
}
```

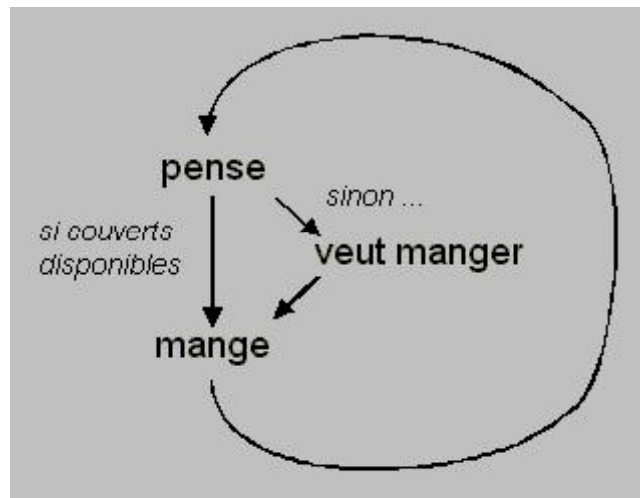
Le dîner des philosophes

- ▶ Cinq philosophes sont réunis autour d'une table pour manger un plat chinois.



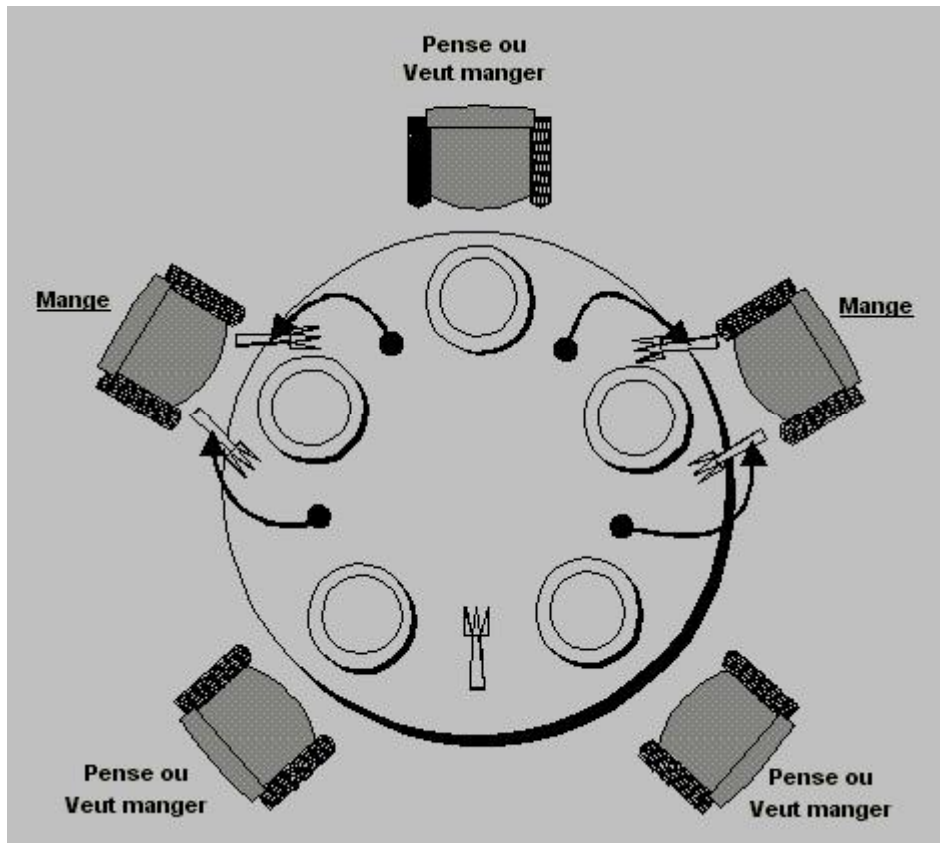
- ▶ Chaque philosophe a devant lui une assiette et un couvert.
- ▶ Un philosophe a besoin de deux couverts pour manger.

Le dîner des philosophes



- ▶ Lorsqu'il ne mange pas, le philosophe réfléchit.

dîner des philosophes à un instant donné



- Penser ou manger
- Deux fourchettes pour manger
- Pas de parole
- Pas d'interblocage
- Pas de famine

Le dîner des philosophes

```
#define N 5                                     /* number of philosophers */

void philosopher(int i)                         /* i: philosopher number, from 0 to 4 */
{
    while (TRUE) {
        think( );                               /* philosopher is thinking */
        take_fork(i);                          /* take left fork */
        take_fork((i+1) % N);                  /* take right fork; % is modulo operator */
        eat( );                                /* yum-yum, spaghetti */
        put_fork(i);                           /* put left fork back on the table */
        put_fork((i+1) % N);                   /* put right fork back on the table */
    }
}
```

Le dîner des philosophes

```
#define N          5          /* number of philosophers */
#define LEFT      (i+N-1)%N   /* number of i's left neighbor */
#define RIGHT     (i+1)%N     /* number of i's right neighbor */
#define THINKING  0          /* philosopher is thinking */
#define HUNGRY    1          /* philosopher is trying to get forks */
#define EATING    2          /* philosopher is eating */
typedef int semaphore;        /* semaphores are a special kind of int */
int state[N];                /* array to keep track of everyone's state */
semaphore mutex = 1;         /* mutual exclusion for critical regions */
semaphore s[N];              /* one semaphore per philosopher */

void philosopher(int i)      /* i: philosopher number, from 0 to N-1 */
{
    while (TRUE) {           /* repeat forever */
        think();             /* philosopher is thinking */
        take_forks(i);       /* acquire two forks or block */
        eat();               /* yum-yum, spaghetti */
        put_forks(i);        /* put both forks back on table */
    }
}
```

Le dîner des philosophes

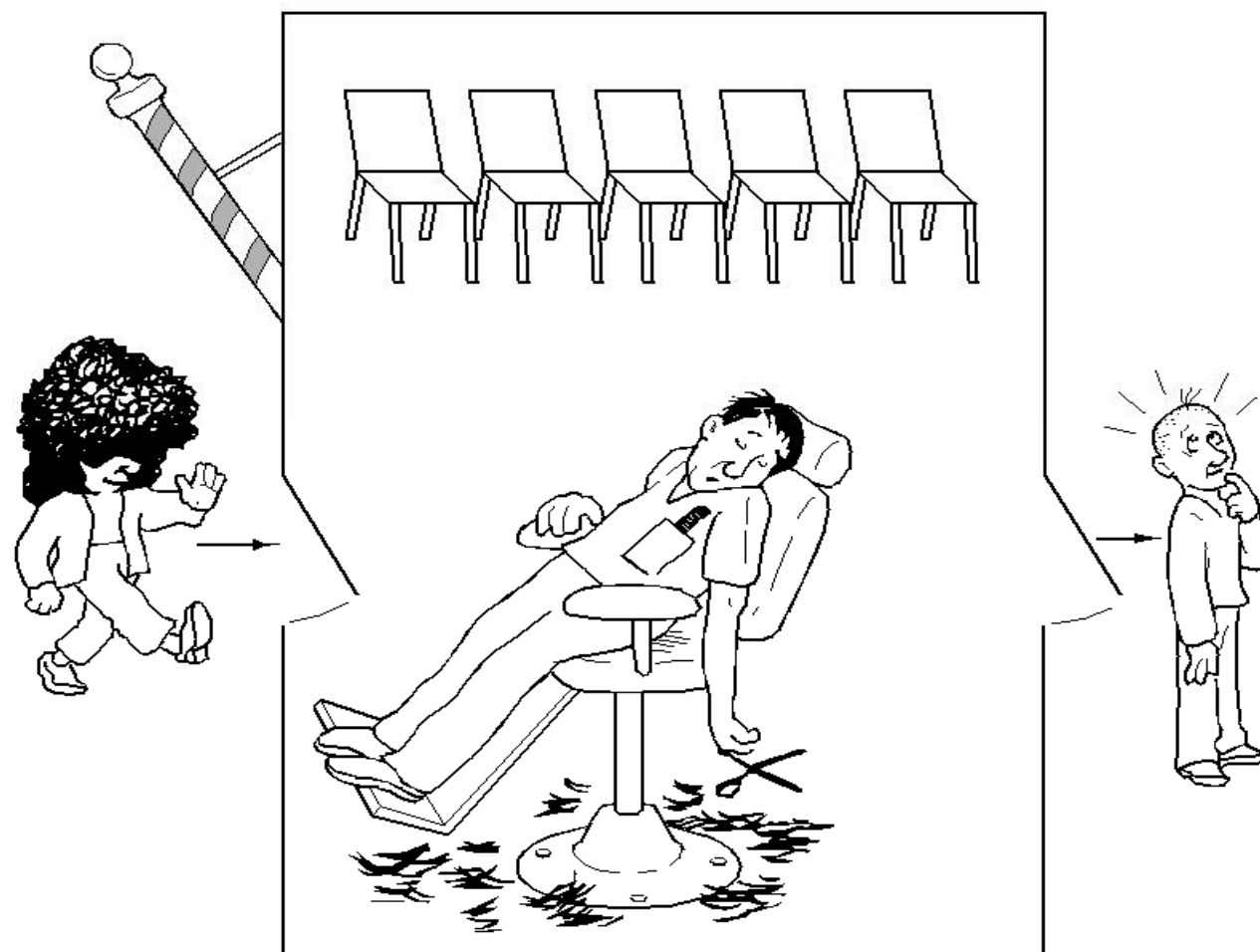
```
void take_forks(int i)                                /* i: philosopher number, from 0 to N-1 */
{
    down(&mutex);                                     /* enter critical region */
    state[i] = HUNGRY;                                /* record fact that philosopher i is hungry */
    test(i);                                           /* try to acquire 2 forks */
    up(&mutex);                                        /* exit critical region */
    down(&s[i]);                                       /* block if forks were not acquired */
}

void put_forks(i)                                     /* i: philosopher number, from 0 to N-1 */
{
    down(&mutex);                                     /* enter critical region */
    state[i] = THINKING;                             /* philosopher has finished eating */
    test(LEFT);                                       /* see if left neighbor can now eat */
    test(RIGHT);                                     /* see if right neighbor can now eat */
    up(&mutex);                                       /* exit critical region */
}
```

Le dîner des philosophes

```
void test(i)                                /* i: philosopher number, from 0 to N-1 */
{
    if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
        state[i] = EATING;
        up(&s[i]);
    }
}
```


Le coiffeur endormi



Le coiffeur endormi

```
#define CHAIRS 5                                /* # chairs for waiting customers */

typedef int semaphore;                          /* use your imagination */

semaphore customers = 0;                        /* # of customers waiting for service */
semaphore barbers = 0;                         /* # of barbers waiting for customers */
semaphore mutex = 1;                           /* for mutual exclusion */
int waiting = 0;                               /* customers are waiting (not being cut) */

void barber(void)
{
    while (TRUE) {
        down(&customers);                      /* go to sleep if # of customers is 0 */
        down(&mutex);                          /* acquire access to 'waiting' */
        waiting = waiting - 1;                 /* decrement count of waiting customers */
        up(&barbers);                          /* one barber is now ready to cut hair */
        up(&mutex);                          /* release 'waiting' */
        cut_hair( );                          /* cut hair (outside critical region) */
    }
}
```


Le coiffeur endormi

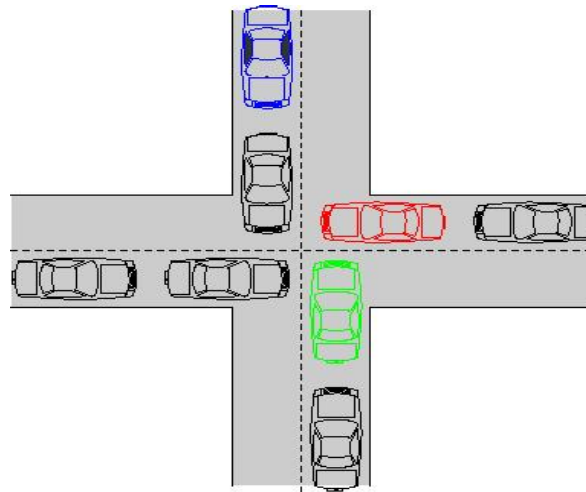
```
void customer(void)
{
    down(&mutex);
    if (waiting < CHAIRS) {
        waiting = waiting + 1;
        up(&customers);
        up(&mutex);
        down(&barbers);
        get_haircut( );
    } else {
        up(&mutex);
    }
}
```

/* enter critical region */
/* if there are no free chairs, leave */
/* increment count of waiting customers */
/* wake up barber if necessary */
/* release access to 'waiting' */
/* go to sleep if # of free barbers is 0 */
/* be seated and be serviced */

/* shop is full; do not wait */

Exemples Introductifs

- ✓ *Exemple 1 :* Considérons un croisement où les voitures s'immobilisent mutuellement : aucune voiture ne peut progresser que si on ne prend pas les mesures spéciales.



- ✓ *Exemple 2 :* Sémaphores -- considérons deux processus P1 et P2 :

P1

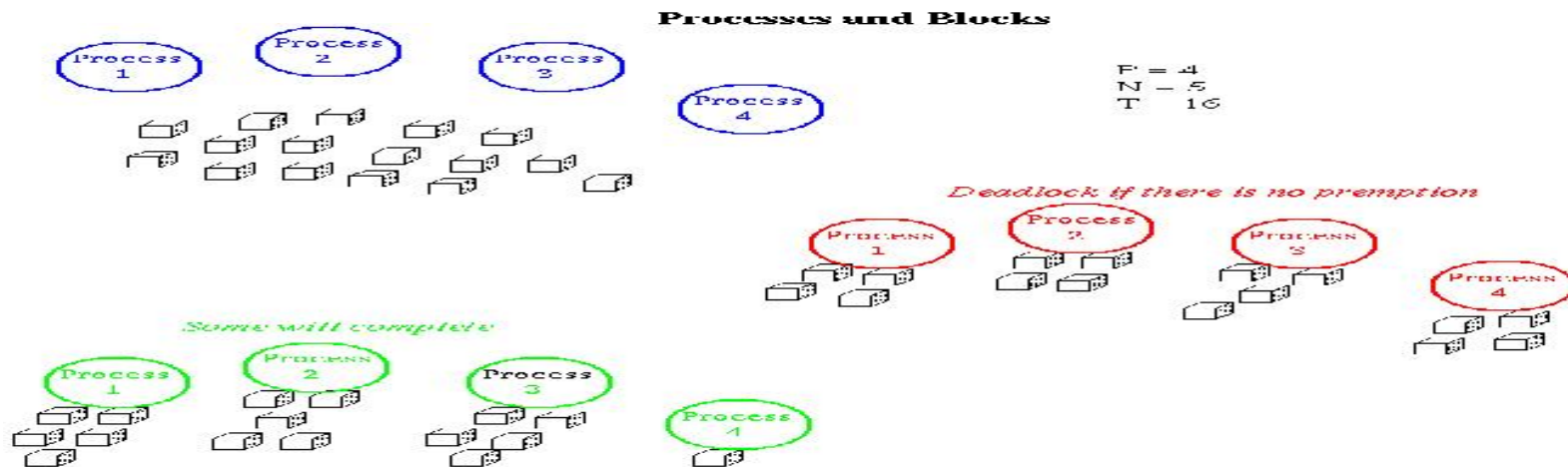
...
Down(**x**)
Down(**y**)
....

P2

....
Down(**y**)
Down(**x**)
....

Exemples Introductifs (suite)

- Les ressources sont, généralement, demandées par des appels système et allouées par le système d'exploitation
- **Définition :** *Le blocage est une situation où chaque processus est en attente d'une ressource détenue par un autre. Puisqu'ils sont tous en attente, aucun ne peut libérer ce que les autres attendent.*
- **Exemple 3 :** Processus et blocs -- Considérons avoir P processus et T blocs. Pour s'exécuter, chaque processus a besoin de N blocs.



Conditions d'apparition du phénomène d'interblocage

La concurrence entre processus et l'allocation dynamique de ressources peuvent introduire un interblocage. Ce phénomène résulte de la conjonction de quatre circonstances :





- *Exclusion mutuelle* : Chaque ressource est utilisable exclusivement par un processus, et un seul, car son partage donnerait des résultats incohérents.
- *Pas de réquisition* : les ressources déjà allouées restent nécessaires aux processus qui les ont reçues et elles ne peuvent être réquisitionnées.

Conditions d'apparition du phénomène d'interblocage (suite)

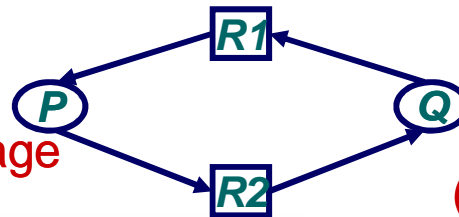
- *Tenir et attendre* : chaque processus ne peut progresser que s'il obtient les ressources qu'il requiert dynamiquement par des demandes successives; il attend donc les ressources de sa dernière requête avant de poursuivre son déroulement.
- *Attente circulaire* : plusieurs processus en attente de ressources allouées à d'autres processus et il s'est formé une attente circulaire



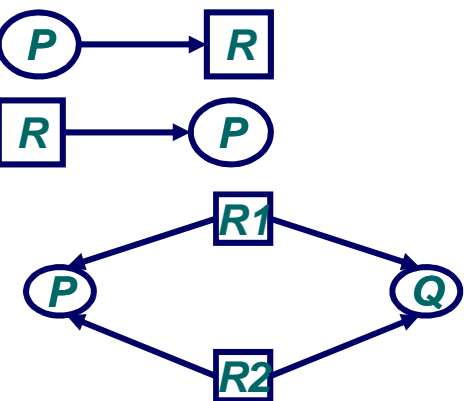
Modélisation des interblocages

- Par un *graphe d'allocation des ressources*, soit $GA(N, A)$ orienté biparti où
 - N : 2 partitions de noeuds
 - un processus P est représenté par 
 - une ressource R est représentée par 
 - A : ensemble des arcs :
 - un processus en attente d'une ressource : 
 - une ressource est allouée à un processus : 
- Exemple d'apparition d'un blocage :

(a) Existence d'un circuit : blocage



(b) Pas d'exclusion mutuelle



Solutions à l'interblocage

1. *Ignorer le blocage* (politique de l'autruche : ça n'arrivera pas)

- Approche adoptée par Unix ("don't worry, it will be fine").
- Si cela arrive, relancer le système !

2. *Détection-Guérison*

- Laisser le blocage se produire puis après l'avoir détecté, reprendre les ressources de certains.
- Rompre le blocage par tuer un ou plus de processus, jusqu'à suppression des circuits (dans le graphe d'allocation des ressources) ; Quelle est la victime?



Solutions à l'interblocage (suite)

3. Méthode préventive

Garantir qu'une situation de blocage ne se produira jamais

Imposer des contraintes sur les demandes de manière que le blocage soit impossible

- Demander toutes les ressources à l'avance pas d'attente
- Exiger de connaître à l'avance les besoins et résultats
- permettre la réquisition
- Numéroté les ressources et les demandes doivent être faites selon l'ordre indiqué

